

Table of Contents

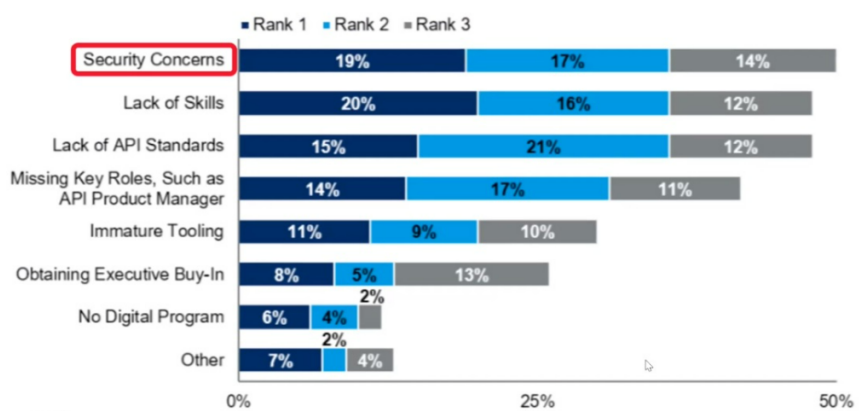
API Security Software Development Life Cycle	3
Different Levels of the API Security Model	3
Best Practices for API security	4
API Ecosystem and common security controls	5
Planning and Requirements	6
Do thread modeling	6
Identify API security requirements	6
Authentication [Requirements] - Who are you? and confirm identity	6
Authorization [Requirements] - What a user is allowed to do? when?	7
Authorization [Techniques]	9
Design	16
Secure architecture - Zero trust, scalable, extensible	16
Sample Code - API architecture, OAuth, Filtering, Claims Principal, Integration tests	18
API Firewalls	24
Encryption sensitive data	26
Infrastructure as code	26
Development	27
Code Review	27
Input Validation	27
Output Encoding	27
Authentication and Authorization mechanisms	27
Proper rate limit and filtering	27
Secure Configuration Management	27
Testing	28
Pen testing	28
Security scanning	28
Vulnerability assessments (outlined below)	28
Security Test Cases	29
Deployment	30
Setup secure pipelines	30
Configuration and testing of secure API firewalls	30
Implement intrusion and detection systems	31
Retirement	32
Testing - Vulnerability Assessments	32
Injection	32
Best Practices	32
Broken User Authentication session management	33
Best Practices	33
Excessive Data Exposure	34
Best Practices	34

Lack of Resources and Rate Limiting	34
Best Practices	34
Broken Object-level Authorization “BOLA”	35
Best Practices	35
Broken Function-Level Authorization	35
Best practices	35
Mass Assignment Vulnerabilities	36
Best Practices:	36
Security Misconfiguration	36
Best Practices	36
Improper Asset Management	37
Best Practices	37
Insufficient Logging and Monitoring	37
Best Practices:	37

API Security Software Development Life Cycle

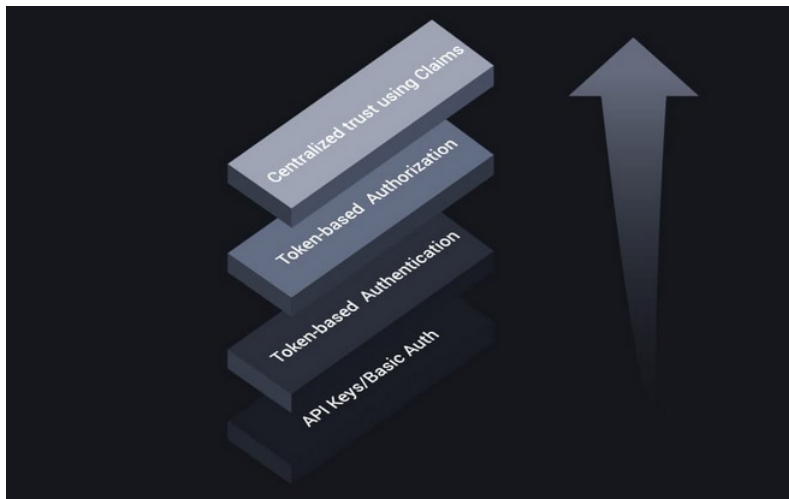
Security is a top API challenge for organizations

OUR CUSTOMERS ARE STILL LEARNING ABOUT APIS



API security encompasses the practices, processes, and products used to ensure APIs are secure, data can be transferred safely, and malicious attacks are prevented

The API Security Maturity Model reframes the model within the context of security



Different Levels of the API Security Model

- Level 0: API Keys and Basic Authentication
- Level 1: Token-Based Authentication
- Level 2: Token-Based Authorization
- Level 3: Centralized Trust Using Claims

APIs that utilize OAuth and OpenID Connect can take advantage of Claims, an advanced form of trust.

Tokens such as JWTs utilizing Subject and Context Attributes can delegate platform-wide trust

Using protocols like OAuth and OpenID Connect, an app can share secure, asserted data within JWTs for verification.

Neo-security Stack



This protocol suite gives us all the capabilities we need to build a secure API platform.

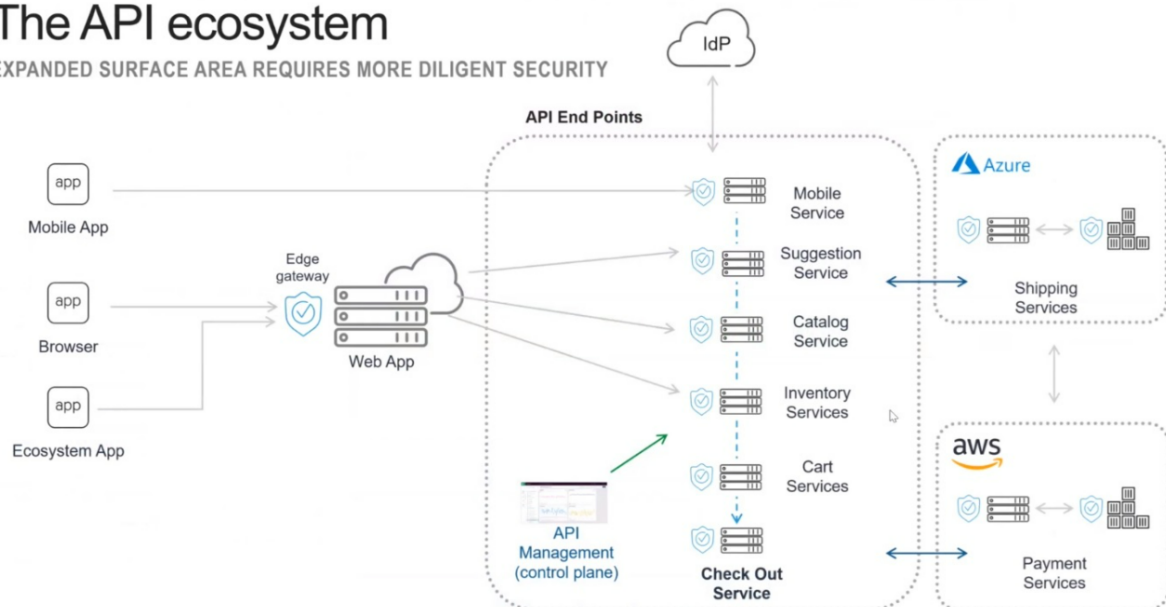
Best Practices for API security

- Proper API Authentication and Authorization
- Zero Trust for APIs
- API Discovery, Monitoring, and Auditing
- Proper Use of Tokens: JWT, Phantom Token, Split Token

API Ecosystem and common security controls

The API ecosystem

EXPANDED SURFACE AREA REQUIRES MORE DILIGENT SECURITY



All APIs need common controls



Use cases drive logical designs



Planning and Requirements

Do thread modeling

Understand the entire business workflow applicable to an API and how it can be attacked.
Identify the weak spots, under the product, workflow and/or pipelines.

Identify API security requirements

Authentication [Requirements] - Who are you? and confirm identity

Questions to revisit:

- What type of roles are available under a secure API:
Administrator, Developer, User, Auditor, Third Party Integrator, Support
Each of these roles would be associated with specific permissions and limitations ensuring that API security is maintained while allowing necessary functionalities to different users and systems
- How to implement these roles? And link - assign them to users?
- Authentication types? Multi-factor authentication is most recommendable path
Single sign on
- Authentication Models:
Access Control List (ACL) - difficult to maintain and scale
Role Based Access Control - simpler - permission assigned to a role - multiple roles can be assigned to a user. **downside** Role explosion inside an org can have more roles than users.
Attribute Based Access Control - ABAC
attributes determine the entitlement of a user to permissions after runtime, attributes for the subject, the resource and the action around the access request is gathered together along with context and then an authorization policy is applied, that yields as a result "authorization decision", based on the combination of attributes and the policy
Policy Based Access Control - PBAC
same as ABAC
- What resources are available and under what circumstances?
Authentication Policy Stakeholders:
 - Application owners
 - Business owners
 - IT
 - App developers
 - Owner of the data (delegating access)
 - Externalize
 - Ability to change authorization without changing code and app

Authorization [Requirements] - What a user is allowed to do? when?

Questions to revisit:

- Authorization Policy (rules) should be breakdown into rules before jump into frameworks or technologies
 - > Actor can approve txn in their own region when the txn amount is under \$\$
- Fine-grained authorization models leverage attributes to compute permissions | entitlements dynamically

How is the type of user determined?

What region is the user in, and what region does the user belongs too

What records are been granted access based on the requesting user and privileges or ownership linked to the records

- What is allowed to do by a given user? Can be explored by revisiting in more detail
 - * **Attributes**
 - * **Tokens**
 - * **Claims**
 - * **Scopes**

Attributes make up - building blocks in conjunction with current policy in place

Attributes are used as inputs for the policy and the policy decision point, dynamically generates an access decision

Attributes are building blocks and serve as input to a policy for fine-grained authorization

Breaking down the attributes

Tellers can approve transactions in their own region when the transaction amount is under \$10,000

A user with the `role == teller` can perform the `action == approve` on resources of `type == transaction` if `transaction.region == user.region` and `transaction.amount < $10,000`

MORE VIDEOS

Tokens

Convenient way to transport the attribute data needed to perform authorization
Tokens can be issued differently. Depends on the client used to request the token and depends on the scope requested. Tokens are bound to a specific application, API, use case. Can be signed or encrypted

Claims are assertions allowing an application or API to trust the attributes.
Allows the user to consent what information or data shares with an app and API

Scopes are a grouping of claims that define “scopes of access”. Used to define access.
Scopes are string values that are consumed by APIs to grant access to requested operations on requested resources (view accounts for example)
A common way to get started with scopes is to use a combination of the type of resource and the access required on it.

Access Level	Resource Type	and	Scope Value
Read	order		read_order



Authorization [Techniques]

Topics to revisit:

Claim Mappers & Value Providers

Authorization in Layers

Entitlement Management System

Attribute Governance

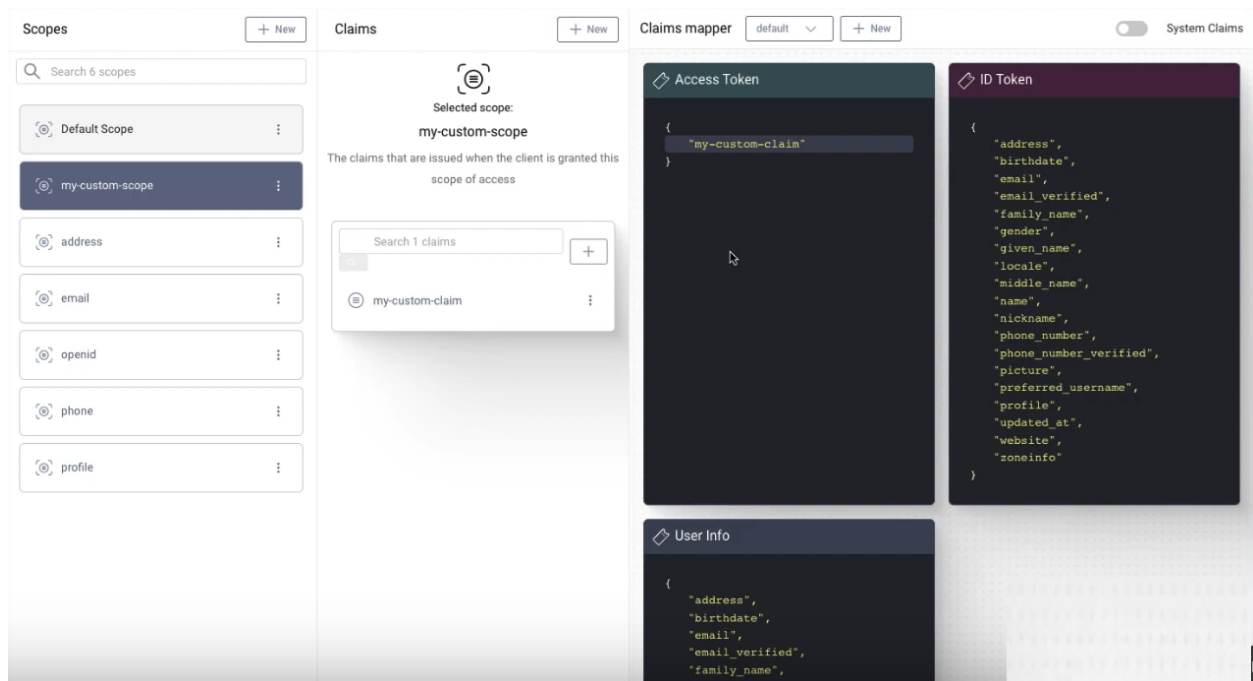
Auditing

→ Claim Mappers

Defines the way that claims are mapped to tokens

Mapper controls what claims are allowed to be included in a token

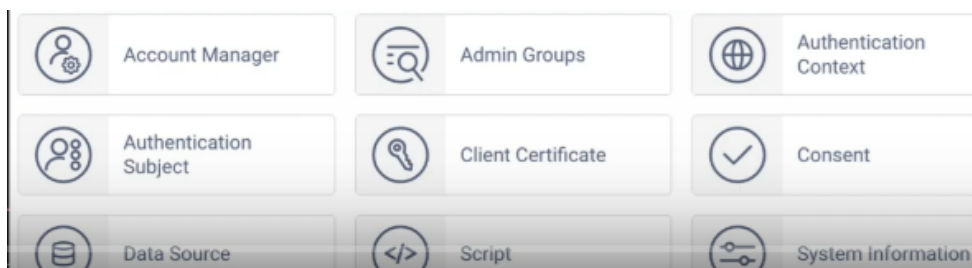
Different mappers can be assigned to different clients

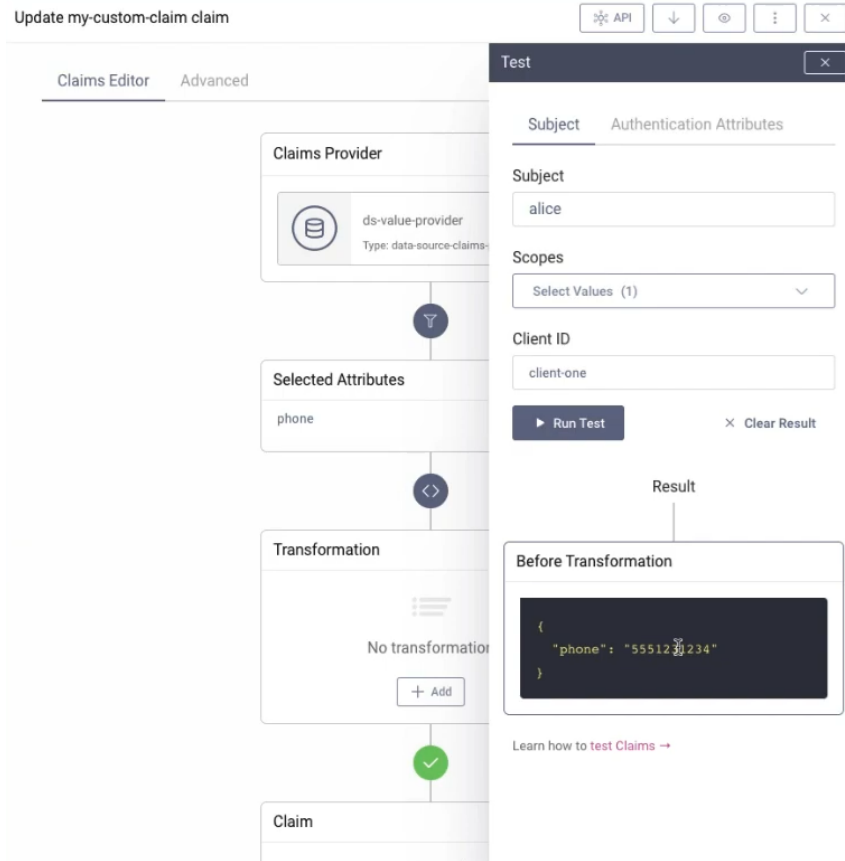


→ Claim Value Providers

Connect to any type of data source to retrieve data specific to the domain / app / API

Intend to fetch attribute (claim) values to be used for authorization



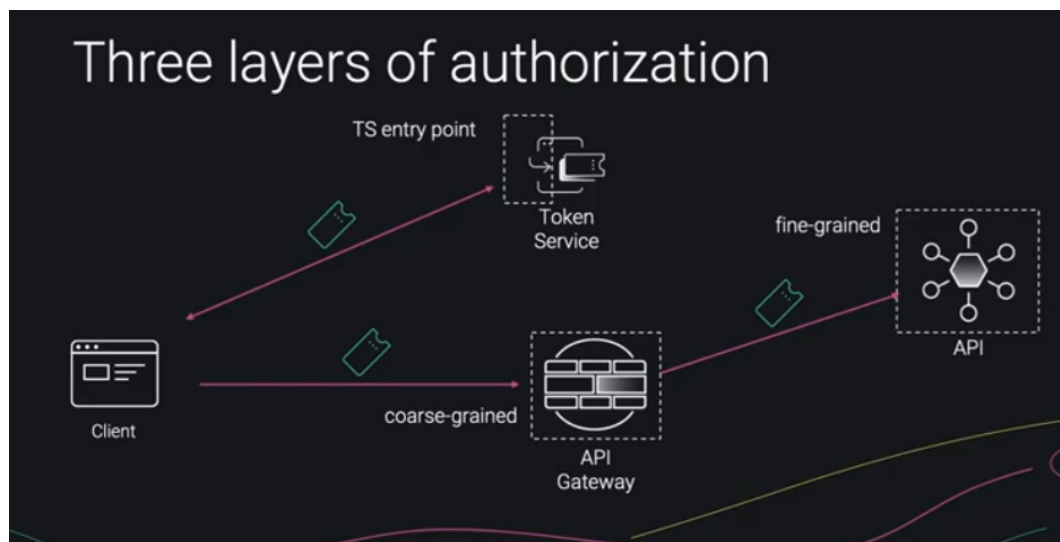


→ Authorization in Layers

Token service - what scopes/claims are allowed to be requested?

API Gateway - coarse grained check of scopes

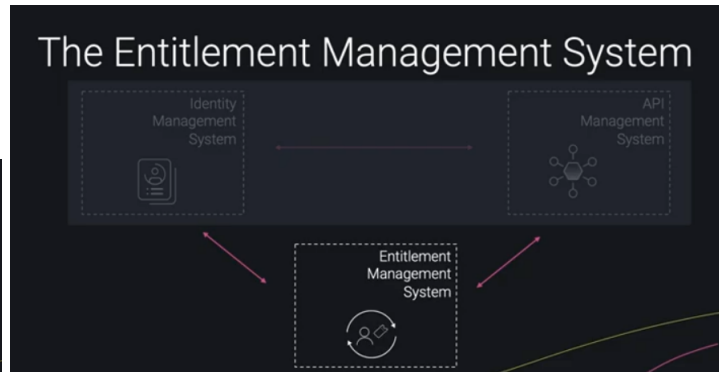
API - fine grained authorization of data released by API. Check claims, issues, and audience.



If you do not have the correct scope, then the API doesn't even make any requests.

→ Entitlement Management System

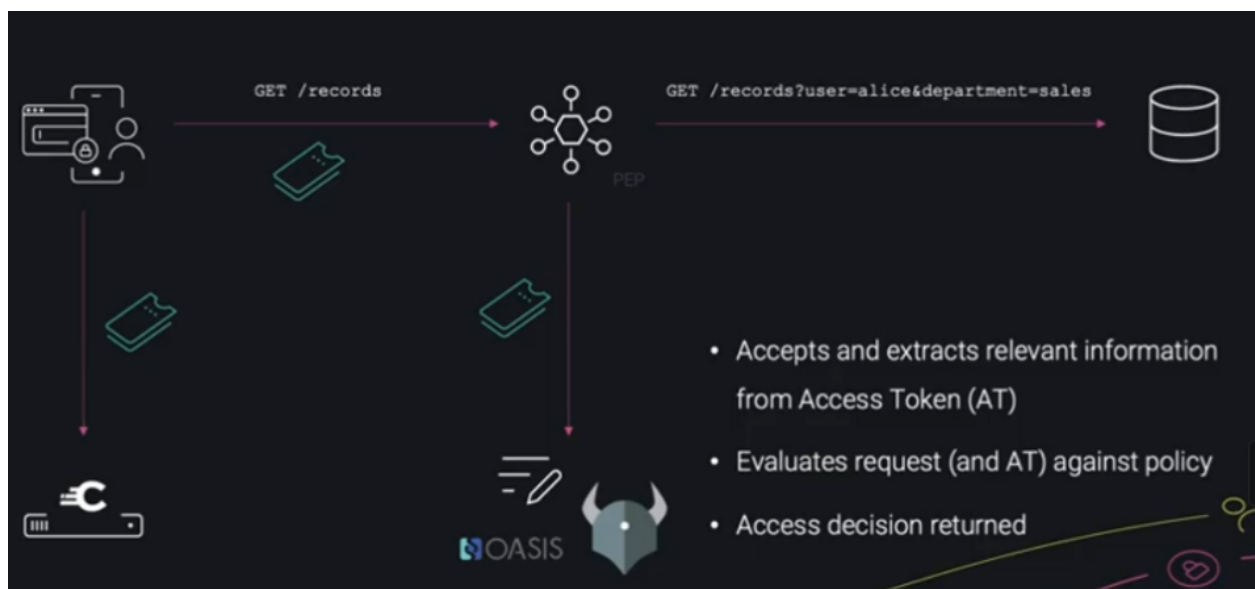
Component within the architecture that can be used to externalize the policy that defines what access is granted under what circumstances. Relies on configuration of policies and rules within the component



Internal components within - Entitlement Management System



- ### Entitlement Management System
- Extract and centralize policy
 - Used by different components across multiple layers
 - Cleaner application code and easier to maintain
 - Central place for policy audit



EXAMPLE of a policy

package records

default allow = false

only until ALL below are TRUE, then the request will be performed

```
allow {  
  is_issuer  
  is_get  
  is_records  
  is_aud  
  match_claims  
  is_owner  
}
```

Allowed issuer(s)

```
issuers = {"https://opa-kong-idsvr:8443/oauth/v2/oauth-anonymous"}
```

```
is_issuer {  
  input.token.payload.iss == issuers[issuer]  
}
```

```
is_get {  
  input.method == "GET"  
}
```

```
is_records {  
  startswith(input.path, "/api/records")  
}
```

```
is_aud {  
  input.token.payload.aud = "www"  
}
```

```
match_claims {  
  input.token.payload.scope = "openid records"  
}
```

(*) Calls record_owner method to verify if the request record is owned by the requesting user (sub)

```
is_owner{  
  record_id := trim_left(input.path, "/api/records/") ##trim the path to get record id  
  lower(input.token.payload.sub) == lower(record_owner(record_id).patient)  
}
```

Method takes a record_id as input to resolve record data

```
record_owner(record_id) = http.send({  
  "url": concat("", ["http://opa-kong-tutorial-api:8080/api/records/", record_id]),  
  "method": "GET",  
  "force_cache": true,  
  "force_cache_duration_seconds": 86400 # Cache response for 24 hours  
}).body
```

(*) Requestor of the token needs to be the same as the patient of the record that your are trying to access.

→ Attribute Governance

Attribute data used for authorization decision

What is the expected flow if an attribute value changed?

Governance needs to be applied to control who, how, where and when an attribute data can be changed?

→ Auditing

What access decision were made

Log data

Snapshot - if attributes change, current access might be different

Who had access to what and under what circumstances?

Query the PDP with appropriate attributes to test the policy

Notes:

Policy Governance and Attribute Governance, when attributes change, can be important due privileges might be different for the future.

Running tests towards entitlement management and towards policy and attributes and corroborate the resulting decisions.

Identify the appropriate types authentication flows

Basic Setup ✕

License Database for tokens Database for accounts Email provider SMS provider SSL Keys OAuth profile Done

Create Authentication and Token Profiles.

Create a default set of authentication and token profiles.

Web and Mobile flows

☒ Code Flow ☒ Enable OpenID Connect

☒ Implicit Flow ☒ Expose Metadata

☒ Assisted Token Flow

☒ Resource Owner Credentials Flow

☒ Token Exchange Flow

☐ Back-channel Authentication Flow

API and Backend flows

☒ Client Credentials Flow

☒ Introspect Flow





Previous Next

Identify the appropriate or target clients?

Clients

Find clients by ID, name, or capabilities

+ New Client

Logo	ID	Name	Capabilities	Action
	gateway-client			<div><div>⚙ Edit</div><div>▼</div></div>
	www			<div><div>⚙ Edit</div><div>▼</div></div>

Identify the appropriate type of scopes?

Code Flow

Enter client ID, select environment and run a new code flow.

Configure your client with the following redirect URI:

`app://oauth.tools/callback/code`

1 Settings

☐ Use PKCE

▼ OPENID SETTINGS

Start Flow

Start URL 

```
https://idsvr:8443/oauth/v2/oauth-authorize?
&client_id=www
&state=1623257237962-s4S
&scope=openid%20records
&response_type=code
&redirect_uri=app://oauth.tools/callback/code
```

Ensure a valid access token is generated

Clear result

Show guide

Token 1

ACCESS TOKEN

226c73b3-dd0c-422c-b31e-c8f043dbcb2e

Token 2

REFRESH TOKEN

9fd6dd7f-36d2-4796-8bbd-8ffcc9b10c20

Token 3

ID TOKEN

Perform an API call

Call API

Select a token from an existing collection or type a new one to call an external API endpoint.

Settings

http://localhost:8000/records/2

GET

Type a token or select one from a Collection

Send

Headers

Query Parameters

Header Name

Header Value

Request preview in curl

```
curl -sS -X GET \
http://localhost:8000/records/2 \
-H 'Authorization: Bearer fe9292f4-fa14-4292-ae21-ca667a9c02c3'
```

Confirm access to the records - appropriately

Call API

Select a token from an existing collection or type a new one to call an external API endpoint.

Settings

http://localhost:8000/records/0

GET Code Flow: Access Token 226c73b3-dd0c-422c-b... Send

Headers Query Parameters

Header Name Header Value

Request preview in curl

```
curl -Ss -X GET \
http://localhost:8000/records/0 \
-H 'Authorization: Bearer 226c73b3-dd0c-422c-b31e-c8f043dbcb2e'
```

HTTP 200 success

Response Body

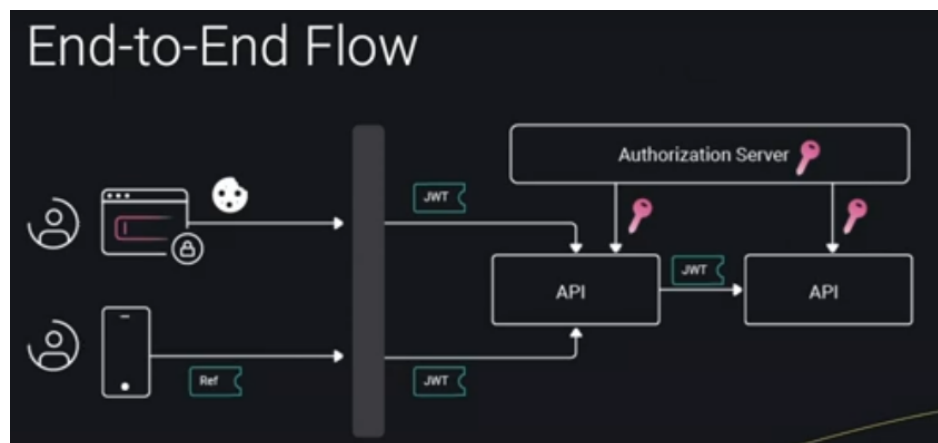
```
{
  id: 0,
  patient: Alice,
  doctor: Karen,
  region: USA,
  notes: Lorem ipsum dolor sit amet, consectetur.
}
```

Response Headers

```
content-type: application/json
content-length: 110
connection: close
date: Thu, 17 Jun 2021 18:19:12 GMT
x-kong-upstream-latency: 3
x-kong-proxy-latency: 1279
via: kong/2.4.1
```

Design

Secure architecture - Zero trust, scalable, extensible



- [Scalable] Separate API from Authentication and client permissions
- Emphasize the importance of an identity focus in API
- Explicit and continuous verification
- Least privilege access
- Assume a breach occurs, how to minimize impact?

- API security with OAuth and JWT validation

API can use an OAuth filter to implement its JWT validation on every request.

API then uses claims to implement its business authorization.

API's integration tests can use libraries to create JSON Web Keys to enable tests to productively issue mock access tokens as any user.

A JWKS endpoint is spun up to expose the mock token signing public key. Mock access tokens have the same contract as real access tokens.

During integration tests, the API makes all the correct OAuth security checks before the API call

>> Design Requirements

- Identify user and client authentication
- Determine JWT access tokens as API message credentials
- Identify scopes and claims for authorization
- Design for simple code - scalable to many APIs

Validate JWT

Check scopes - fixed at design time

Use Claims - dynamically calculated at run time

>> Data Protection Requirements in APIs

- Define permissions for clients and users
- Identify business authorization rules
- Outline trust relationships

>> Design the Claims Principal

- Identify Business Domain Claims
- Validate Claims are retrieved during token issuance
- Test Claims are able to return appropriate authorization decision as expected



>> Token Delivery Design Patterns

- Use an API gateway to manage differences in API credentials
- Token Handler Pattern - for cookies
- Phantom Token Pattern - to introspect opaque tokens, leverage API gateway.

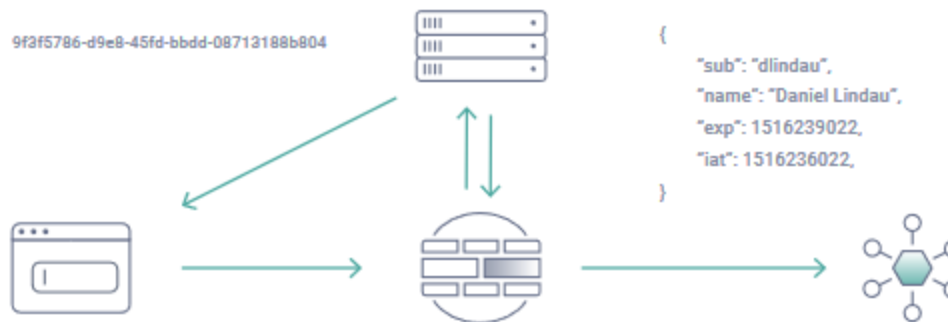
the server issues an opaque access token, a token that is merely a reference to the token data. The opaque token is represented by a random string, so there is no way for

a

potential attacker to extract any data from it. Understood by API only.

With the API gateway in place, it is possible to perform the introspection for the API

Phantom Token



>> Recommended Best Practices

- JWT access tokens are meant for API
- Return to opaque tokens to clients, to avoid information disclosure
- Return secure HTTP-only cookies
- Organize sensitive data only to be reachable by OIDC server
- Include identity data in token, not context attributes
- Limit data exposure only to when the client needs it
- Avoid app-specific rules

Sample Code - API architecture, OAuth, Filtering, Claims Principal, Integration tests

```
/**
 * An OAuth filter to do JWT validation
 */
public class OAuthFilter implements Filter {

    public static final String CLAIMS_PRINCIPAL = "CLAIMS_PRINCIPAL";
    private static final Logger _logger = LoggerFactory.getLogger(OAuthFilter.class);
    private final ServerOptions _options;
    private final HttpsJwksVerificationKeyResolver _httpsJwksKeyResolver;

    public OAuthFilter(ServerOptions options) {
        _options = options;
        _httpsJwksKeyResolver = new HttpsJwksVerificationKeyResolver(new HttpsJwks(options.getJwksUrl()));
    }

    @Override
    public void init(FilterConfig filterConfig) {
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) throws IOException,
        ServletException {

        var httpRequest = (HttpServletRequest) request;
        var httpResponse = (HttpServletResponse) response;
```

```

try {

    var jwt = this.getBearerToken(httpRequest);
    if (jwt.isEmpty()) {
        _logger.info("No access token was received in the authorization header");
        this.unauthorizedResponse(httpResponse);
        return;
    }

    var jwtConsumer = new JwtConsumerBuilder()
        .setVerificationKeyResolver(_httpsJwksKeyResolver)
        .setJwsAlgorithmConstraints(
            AlgorithmConstraints.ConstraintType.PERMIT,
            AlgorithmIdentifiers.RSA_USING_SHA256
        )
        .setExpectedIssuer(_options.getIssuer())
        .setExpectedAudience(_options.getAudience())
        .build();

    var jwtClaims = jwtConsumer.processToClaims(jwt);

    var scopeString = jwtClaims.getStringClaimValue("scope");
    var scopes = scopeString.split(" ");
    var foundScope = Arrays.stream(scopes).filter(s -> s.contains(_options.getScope())).findFirst();
    if (foundScope.isEmpty()) {
        _logger.info("The JWT access token has an invalid scope");
        this.forbiddenResponse(httpResponse);
        return;
    }

    _logger.debug("The request passed JWT validation");
    request.setAttribute(CLAIMS_PRINCIPAL, jwtClaims);

    if (filterChain != null) {
        filterChain.doFilter(request, response);
    }

} catch (InvalidJwtException ex) {

    _logger.info("JWT validation failed");
    for (var item : ex.getErrorDetails()) {
        String info = String.format("%s : %s", item.getErrorCode(), item.getErrorMessage());
        _logger.debug(info);
    }

    this.unauthorizedResponse(httpResponse);

} catch (MalformedClaimException ex) {

    _logger.info("The scope could not be found in the JWT access token");
    this.forbiddenResponse(httpResponse);
}
}

```

```

@Override
public void destroy() {
}

private String getBearerToken(HttpServletRequest httpRequest) {

    var header = httpRequest.getHeader("Authorization");
    if (header != null) {
        var parts = header.split(" ");
        if (parts.length == 2) {
            if (parts[0].equalsIgnoreCase("bearer")) {
                return parts[1];
            }
        }
    }

    return "";
}

private void unauthorizedResponse(HttpServletResponse httpResponse) {

    httpResponse.setHeader(
        "www-authenticate",
        "Bearer, error=invalid_token, error_description=Access token is missing, invalid or expired");
    halt(401);
}

private void forbiddenResponse(HttpServletResponse httpResponse) {
    halt(403);
}
}

/**
 * Server were the API makes OAuth filter security checks before an actual API call
 */

/** Start the server with the given product service and options.
 * This sets up the routes for the product service and makes sure that the routes are protected by OAuth.
 * @param productService the service that can access the products
 * @param options different options to start the server with
 * @throws ServletException if the oauth filter cannot be initialized
 */
public Example(ProductService productService, @Nullable ServerOptions options) throws ServletException {

    ServerOptions appliedOptions = Objects.requireNonNullElseGet(options, ServerOptions::new);
    port(appliedOptions.getPort());
    init();

    // Run the filter before any api/* route
    Filter oauthFilter = toSparkFilter(new OAuthFilter(options));
    before("/api", oauthFilter);
    before("/api/", oauthFilter);
    before("/api/*", oauthFilter);

    // Set up the product service to respond to /products and /products/productId routes
    path("/api", () ->
        path("/products", () -> {
            get("", new ListProductsRequestHandler(productService));
            get("/:productId", new GetProductRequestHandler(productService));
        })
    );
}

```

/**

**Abstract API Authorization Integration test, where mock access token are leveraged
JWKS endpoint is spun up to expose the mock token signing public key**

*/

```
public abstract class AbstractApiAuthorizationTest {

    static final String ISSUER = "http://localhost:8443/oauth/v2/oauth-anonymous";
    static final String AUDIENCE = "api.example.com";
    static final String JWKS_PATH = "/oauth/v2/oauth-anonymous/jwks";
    static final String SCOPE = "products";
    static final int PORT = 9090;

    private static boolean started = false;

    /**
     * Creates JWTs for the given issuer and a generated key ID
     */
    static MockJwtIssuer mockJwtIssuer = new MockJwtIssuer(ISSUER, UUID.randomUUID().toString());
    /**
     * Used to mock JWKS endpoint for mocked JWT issuer
     */
    static WireMockServer mockAuthorizationServer;

    @BeforeAll
    public static void startMockAuthorizationServer() throws ServletException {

        if (started) {
            return;
        }

        started = true;
        var options = new WireMockConfiguration().port(8443);
        mockAuthorizationServer = new WireMockServer(options);
        mockAuthorizationServer.start();

        String jwksUrl = mockAuthorizationServer.baseUrl() + JWKS_PATH;
        Logger.getLogger(AbstractApiAuthorizationTest.class.getName()).info("Mocked JWKS URL on " + jwksUrl);
        mockAuthorizationServer.stubFor(get(JWKS_PATH)
            .willReturn(
                ok(mockJwtIssuer.getJwks())
            )
        );
    }

    /**
     * Send an authenticated request to the given url
     * @param subjectName name of authenticated user
     * @param claims claim names and values that should be added to the user's token
     * @param url endpoint to send request to
     * @return response from server as string or null if there was an error.
     */
    HttpResponse<String> sendAuthenticatedRequest(String subjectName, Map<String, String> claims, String url) {
        String jwt = mockJwtIssuer.getJwt(subjectName, claims, AUDIENCE);
        return sendRequest(url, jwt);
    }
}
```

```

/**
 * Send an unauthenticated request to the given url
 * @param url endpoint to send request to
 * @return response as received from server
 */
HttpResponse<String> sendUnauthenticatedRequest(String url) {
    return sendRequest(url, null);
}

/**
 * Send a request to the given url and add JWT to authorization header if available
 * @param urlString endpoint to send request to
 * @param jwt optional, token to add to the authorization header
 * @return response from server/endpoint
 */
private HttpResponse<String> sendRequest(String urlString, @Nullable String jwt) {
    try {
        URI uri = new URI(urlString);
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest.Builder httpRequestBuilder = HttpRequest
            .newBuilder()
            .uri(uri)
            .header("accept", "application/json")
            .GET();

        // Add JWT as bearer token if available
        if (jwt != null) {
            httpRequestBuilder.header("Authorization", String.format("Bearer %s", jwt));
        }

        // return response body as string
        return client.send(httpRequestBuilder.build(), HttpResponse.BodyHandlers.ofString());
    } catch (URISyntaxException | IOException | InterruptedException exception) {
        Assertions.fail(String.format("Cannot send request to %s", urlString));
    }
    return null;
}

/**
 * Get the full URL of the given path for the API
 * @param path relative path/file part of URL
 * @return URL including protocol, host, port and path
 */
String applicationUrl(String path) {
    try {
        return new URL("http", "localhost", PORT, path).toString();
    } catch (MalformedURLException e) {
        throw new RuntimeException(e);
    }
}
}

```

```

/**
Perform GET API call, based on the scope access granted to a user (request)
*/
public class GetProductAuthorizationTest extends AbstractApiAuthorizationTest {

    private static Stream<Arguments> provideInputForAuthorizedRequests() {
        return Stream.of(
            Arguments.of("Alice", "se", "trial", "1"),
            Arguments.of("Alice", "se", "trial", "3"),
            Arguments.of("Alice", "se", "premium", "5"),
            Arguments.of("Bob", "us", "trial", "1"),
            Arguments.of("Bob", "us", "premium", "2"),
            Arguments.of("Clara", "de", "trial", "4"),
            Arguments.of("Clara", "de", "premium", "4")
        );
    }

    /**
     * Test that users can access products they are authorized to access
     * @param name name of the user
     * @param country country the user is located in
     * @param subscriptionLevel level of subscription the user signed up for
     * @param productId id of the product the user aims to access
     */
    @ParameterizedTest
    @MethodSource("provideInputForAuthorizedRequests")
    void returnJsonObjectForProductWhenUserIsAuthorized(String name, String country, String subscriptionLevel, String
productId) {
        HttpResponse<String> response = sendAuthenticatedRequest(
            name,
            Map.of("country", country,
                "subscription_level", subscriptionLevel,
                "scope", SCOPE),
            applicationUrl("/api/products/" + productId));
        assertEquals(200, response.statusCode(), "Response Code");

        Product product = new ProductServiceMapImpl().getProduct(productId);
        assertEquals(JsonUtil.toJsonObjectWithDescription(product).toString(), response.body());
    }

    @Test
    void returnNotFoundWhenUserIsAuthorizedButProductDoesNotExist() {
        HttpResponse<String> response = sendAuthenticatedRequest(
            "Alice",
            Map.of("country", "se",
                "subscription_level", "trial",
                "scope", SCOPE),
            applicationUrl("/api/products/-1"));
        assertEquals(404, response.statusCode(), "Response Code");
    }

    @Test
    void denyAccessWhenUserIsNotAuthorizedToAccessExclusiveProduct() {
        HttpResponse<String> response = sendAuthenticatedRequest(
            "Alice",
            Map.of("country", "se",
                "subscription_level", "trial",
                "scope", SCOPE),
            applicationUrl("/api/products/5"));
        assertEquals(403, response.statusCode(), "Response Code");
    }
}

```

```

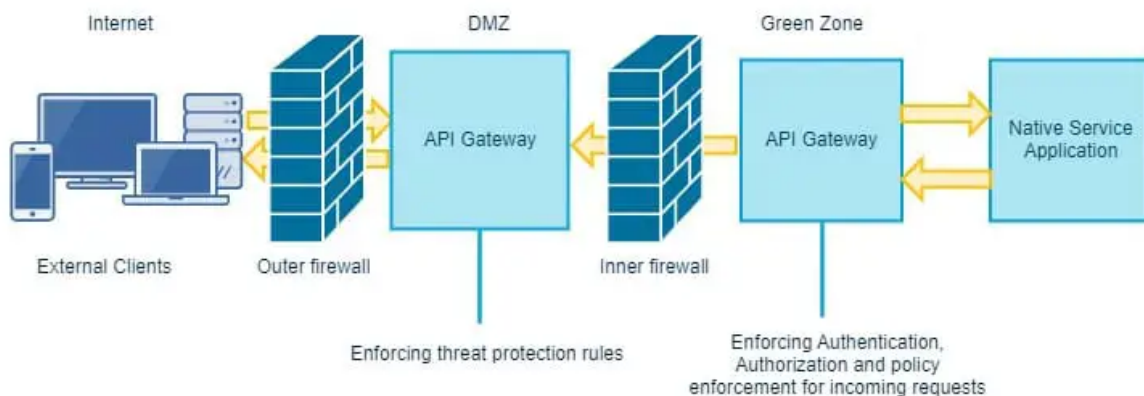
@Test
void denyAccessWhenSubscriptionLevelsEmpty() {
    HttpResponse<String> response = sendAuthenticatedRequest(
        "Alice",
        Map.of("country", "se",
            "subscription_level", "",
            "scope", SCOPE),
        applicationUrl("/api/products/5"));
    assertEquals(403, response.statusCode(), "Response Code");
}

@Test
void denyAccessWhenNoSubscriptionLevel() {
    HttpResponse<String> response = sendAuthenticatedRequest(
        "Alice",
        Map.of("country", "se",
            "scope", SCOPE),
        applicationUrl("/api/products/5"));
    assertEquals(403, response.statusCode(), "Response Code");
}

@Test
void denyAccessWhenUserLoadsProductFromDifferentCountry() {
    HttpResponse<String> response = sendAuthenticatedRequest(
        "Bob",
        Map.of("country", "us",
            "subscription_level", "trial",
            "scope", SCOPE),
        applicationUrl("/api/products/5"));
    assertEquals(403, response.statusCode(), "Response Code");
}
}

```

API Firewalls



API Firewall automatically enforces the API contract spelled out in your API definition.

Each API should have an allowlist of the valid operations and input data based on the API contract, and

API Firewall enforces this configuration to all transactions, incoming requests as well as outgoing responses.

API Firewall:

- * filters out unwanted requests
- only lets through requests that should be allowed based on the OpenAPI definition of the API it protects
- * blocks any responses from the API that have not been declared or that do not match the API definition.

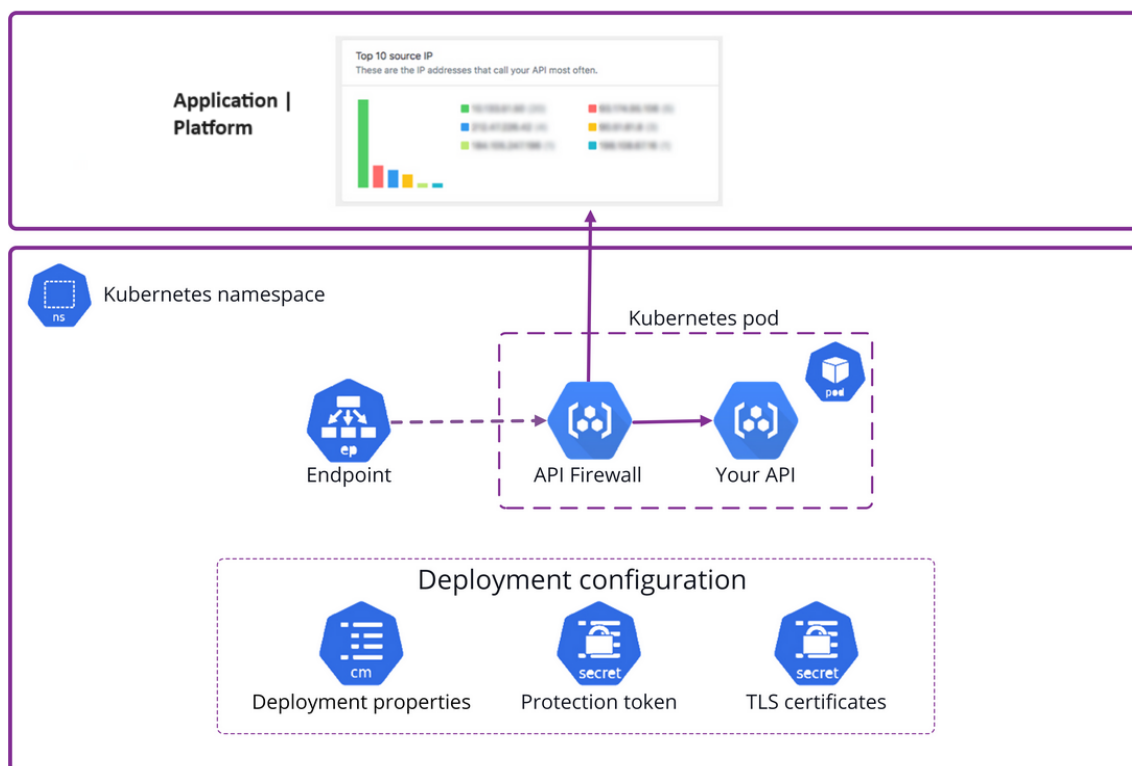
Variables are required to be configured for **the deployment properties** of API Firewall

How they are passed to API Firewall depends on the environment of the API Firewall deployment.

Protection tokens tie protection configurations you create for your APIs to the running API Firewall instances. Protection tokens enable you to secure your API in multiple environments simultaneously. This way you can manage the different cloud deployments for a single API independently of one another.

TLS secrets contain the **TLS certificates** and the corresponding keys for TLS configuration. This should be found in the API variables. Both the certificate file and the private key file must be in PEM format. The TLS configuration files must be in the file system of the API Firewall container.

Sample of a firewall API deployment



Encryption sensitive data

- use HTTPS instead of HTTP to encrypt data in transit
- apply hashing. Hashing is the process of generating a unique and fixed-length value from the data that cannot be reversed.
- using standard encryption algorithms like AES (Advanced Encryption Standard) or RSA, SHA, or HMAC, and store the keys and the hashes securely
- ensure data conforms to expected formats, ranges, and types
- use field-level encryption for particularly sensitive information like passwords or personal identifiers, even if stored in databases.
- ensure that error messages do not reveal sensitive information
- implement rate limiting to prevent abuse and potential DDoS (Distributed Denial of Service) attacks
- monitor and audit can help you detect and prevent any anomalies, errors, or attacks on your API

Infrastructure as code

Managing and provisioning infrastructure through code, it can be leveraged for API security as follows:

- same as code, infrastructure can be version-controlled, reviewed and audited. Ensure that changes are tracked and the entire infrastructure setup is well documented and transparent.
- creation of standardized security patterns that can be reused across multiple APIs and environments. Ensure consistency across security configurations, reducing misconfigurations and vulnerabilities.
- enable the integration of automated security and compliance checks into deployment pipeline, leverage security scanning tools to automatically validate that the infrastructure complies with security policies before it is deployed.
- rapid response and remediation, since its IaC, it can quickly be adjusted and redeployed to address the issue.
- help enforce least privilege access controls for APIs and the underlying infrastructure, by defining the necessary permissions in the code
- adopt an immutable infrastructure approach, where servers are never modified after they are deployed. If a change is needed, a new server is deployed with the change
- easier to scale the infrastructure up or down based on demand, without compromising security. Security configurations and policies scale along with the infrastructure
- integrating monitoring tools into the IaC process allows for continuous monitoring of the infrastructure and APIs

Development

Code Review

- Use static tools to assess vulnerabilities
- Apply secure code guidelines, towards minimize vulnerabilities

Input Validation

- Validate and sanitize user inputs to prevent injection

Output Encoding

- Data is displayed properly to prevent cross script injections XSS

Authentication and Authorization mechanisms

- Apply proper controls to limit access to sensitive resources

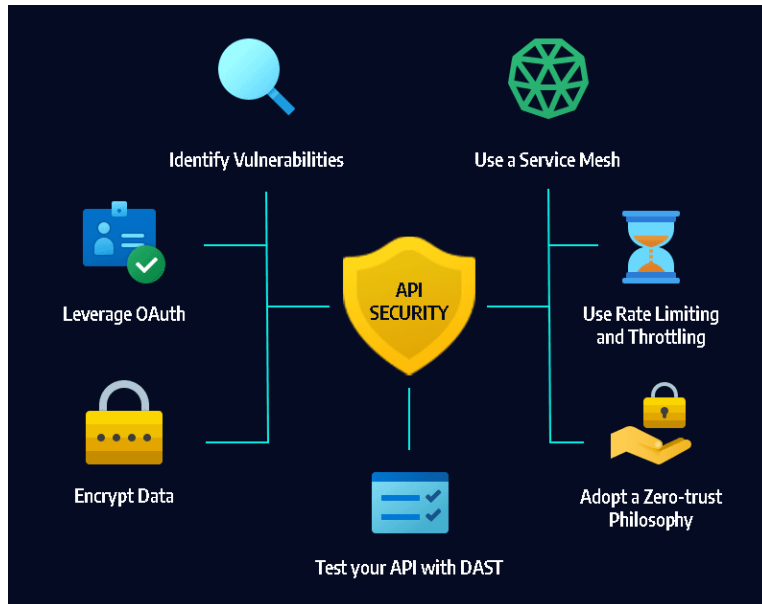
Proper rate limit and filtering

- Setting a cap on how many requests a user can make within a certain time frame. Can be implemented at various levels : IP address, user account, or API token.
- Filtering requests based on IP addresses can help block known malicious sources (blacklisting) or allow only trusted sources (whitelisting).
- Issuing API keys and monitoring their usage helps in tracking and controlling who accesses the API. You can set different permission levels and rate limits for different keys.
- endpoint filtering, can help apply rate limiting and security measures on specific endpoints, especially those that are more sensitive or prone to abuse
- limiting API access based on geographic location can help mitigate risks from regions that are sources of frequent attacks.
- reduce service availability to an individual user as they approach their rate limit, rather than cutting them off abruptly.
- enforce limits on the amount of data or number of resources that can be requested to prevent system overload.
- Use HTTP headers to implement security controls such as Cross-Origin Resource Sharing (CORS) policies.

Secure Configuration Management

- Protect and secure sensitive configuration settings, db user/pwd, API keys, secrets.

Testing



Pen testing

- simulate real world attacks

API security pen testing typically focuses on areas like authentication, authorization, input validation, session management, and API business logic.

Tools commonly used include Postman for API testing, Burp Suite for security testing, and OWASP ZAP (Zed Attack Proxy) for finding vulnerabilities.

Security scanning

- use automated tools to find and prevent vulnerabilities

- * **Static Application Security Testing (SAST)**

evaluates the source code, identifying vulnerabilities.

- * **Dynamic Application Security Testing (DAST)**

evaluates the api while running, and identifies compliance and security problems

- * **Runtime Application Self Protection (RASP)**

analyzes traffic and user behavior while the api is running. These tests can send alerts and respond automatically.

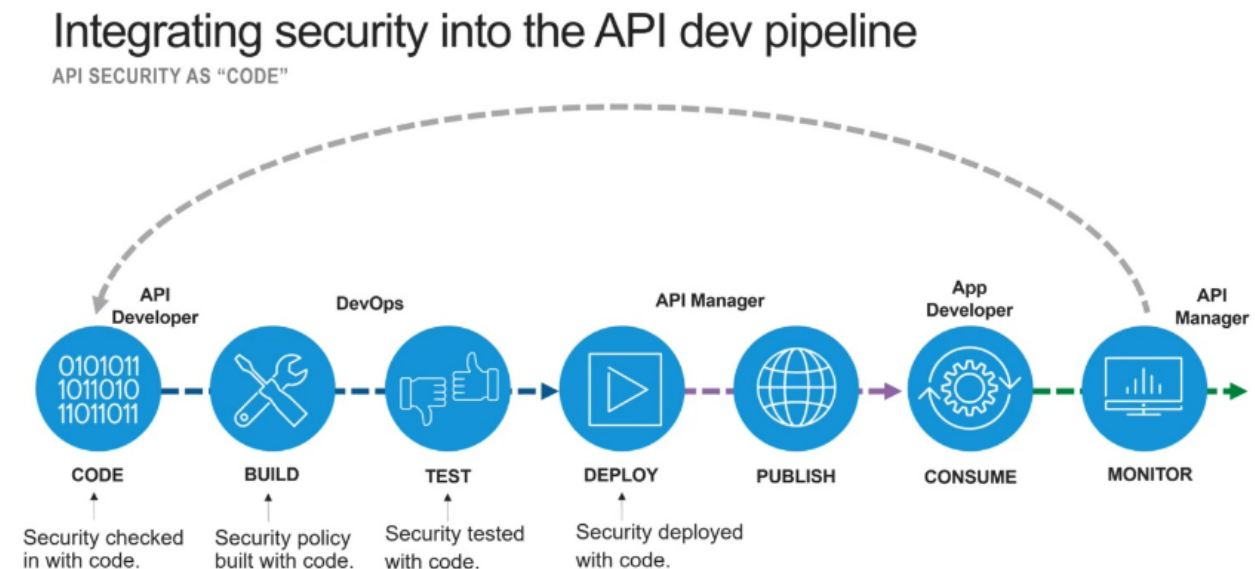
Vulnerability assessments (outlined below)

Security Test Cases

1. **Authentication Tests:**
 - Test with invalid credentials to ensure access is denied
 - Test with expired or revoked tokens
 - Attempt access without authentication
2. **Authorization Tests:**
 - Try accessing resources that should not be accessible due auth user role
 - Test for horizontal and vertical privilege escalation
3. **Input Validation:**
 - Send unexpected data types (e.g., strings in numeric fields)
 - Test with SQL, XML, or script injections
 - Test for buffer overflow vulnerabilities
4. **Data Leakage:**
 - Ensure sensitive data like tokens, passwords are not logged or exposed in responses.
 - Check for excessive error details in API responses.
5. **Session Management:**
 - Test for session fixation attacks
 - Ensure sessions expire after a reasonable time or after logout
6. **Rate Limiting and Throttling:**
 - Test for DoS vulnerabilities by sending numerous requests
 - Check if the system locks out or slows down after a threshold
7. **Cross-Origin Resource Sharing (CORS):**
 - Verify that CORS policy is not too permissive
 - Test with unauthorized domains to ensure they cannot access resources
8. **Data Encryption:**
 - Ensure data in transit is encrypted (using HTTPS)
 - Check for weak encryption algorithms or misconfigurations
9. **Business Logic:**
 - Test for business logic vulnerabilities that could be exploited
 - Attempt to bypass workflow or process steps
10. **Third-Party Libraries and Dependencies:**
 - Check for vulnerabilities in external libraries used by the API
 - Ensure all dependencies are up-to-date and patched
11. **Mobile-Specific Tests:**
 - Test for secure communication between mobile apps and the API.
 - Check for hard-coded credentials or keys in the mobile app.
12. **Cloud-Specific Tests:**
 - Ensure proper configuration of cloud services used by the API.
 - Test for misconfigurations in cloud storage, IAM roles, etc.
13. **Logging and Monitoring:**
 - Ensure all access attempts are logged.
 - Test if the monitoring system alerts on suspicious activities.

Deployment

Setup secure pipelines



* Perform dependency and/or container scanning

Determine where your dependencies are, ensure that versions are used consistently and that all dependencies are up to date.

Container scanning, helps with identify vulnerable configurations, malware infections, insufficient secrets management, and compliance breaches

* Protect endpoints, thru deployment of an endpoint protection platform, including next-generation antivirus to protect against unknown and zero-day malware

Configuration and testing of secure API firewalls

For configuration details - please refer to section Design > API Firewalls in page 4

Testing an API Firewall, some recommendations are:

General

- Ensure the firewall is configured according to best practices.
- Check if the firewall rules are appropriate for the API's specific use case.
- Verify that policies are updated to counter recent vulnerabilities and threats.
- Pen Testing, simulate cyber attacks on the API to see if the firewall can effectively block them.
- Test for various attack vectors, including those specific to APIs like broken authentication, excessive data exposure, etc.
- Verify that the firewall correctly enforces authentication and authorization policies.

- Test with various credentials, including invalid and expired ones, to ensure unauthorized access is blocked.
- Check if the firewall effectively limits the rate of requests to prevent Denial-of-Service (DoS) attacks.
- Test how the firewall handles sudden spikes in traffic.

Logging and Monitoring

- Ensure that the firewall logs all attempts to access the API.
- Review logs to see if any attacks or unusual activities were correctly identified and logged.

Compliance Checks

- If applicable, ensure that the firewall's configuration complies with relevant regulations and standards like GDPR etc.

Updates and Patch Management

- Verify that the firewall is regularly updated with the latest security patches and updates.

Performance Testing

- Assess the impact of the firewall on API performance. It should not introduce significant latency.

User Feedback and Incident Analysis

- Collect feedback from users on any security-related issues they face.
- Analyze past security incidents, if any, to check if the firewall responded appropriately.

Implement intrusion and detection systems

Continuously monitoring API endpoints, by logging all API requests and responses to track who is accessing the API, what resources they are requesting, and how the system is responding.

Integrating threat intelligence feeds to stay updated on the latest API security threats and vulnerabilities. This helps in fine-tuning the detection mechanisms to recognize new types of attacks.

Adhering to security policies and standards like OWASP (Open Web Application Security Project) guidelines, which provide a framework for securing web applications and APIs. Have an incident response plan in place to respond when intrusions detected. This includes isolating affected systems, analyzing the breach, and taking steps to prevent future incidents.

Retirement

When the API is no longer needed or being replaced, it should be decommissioned securely to ensure that any sensitive data is not accessible and that it doesn't become a security liability.

Testing - Vulnerability Assessments

Injection

Vulnerability definition:

Injection of malicious code or data into an API

Impact: It can result in unauthorized access to API sensitive data or functionality

How-to-exploit:

- [1] attackers can exploit input fields in APIs where inputs are not properly sanitized
- [2] API doesn't rigorously validate input data for type, length, format, and range, attackers can send unexpected input to manipulate the backend systems
- [3] API has weak authentication mechanisms
- [4] IDOR when an API exposes a reference to an internal implementation object, like a file, directory, or database key
- [5] APIs don't implement rate limiting, an attacker can overload the API with a high volume of requests

Best Practices

- Parameterized queries
- White-list input validation
- API rate limiting
- Web Application Firewalls
- Limit database privileges
- Regular security audits
- Secure authentication and authorization
- Content type validation
- API security testing

Broken User Authentication | session management

Vulnerability definition:

Enable attackers to impersonate valid users

Impact: Compromise data privacy and infrastructure.

How-to-exploit:

[1] application does not enforce strong password policies or multifactor authentication, or flawed password recovery

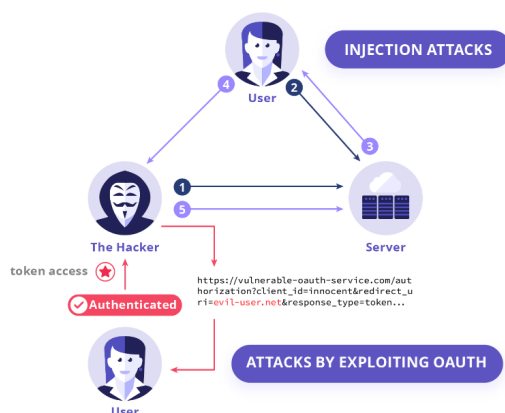
[2] Insecure handling of session tokens (like session cookies) can lead to vulnerabilities

[3] insecure data storage or transmission, unencrypted connections for data transmission

[4] Failing to expire sessions after a user logs out or after a certain period of inactivity

Best Practices

- Implement two-factor authentication
- Secure session management
- Enforce strict password policies
- Implement access control and authorization
- Regularly review session logs
- Token-Based, Attribute, Claim Authentication
- Account lockouts and brute-force protection
- User account monitoring
- Security incident response



1. An attacker introduces malicious code into a server
2. A victim accesses the compromised server and gives input
3. Server returns page code with injected script
4. Victim browser executes script and sends session cookies to attacker
5. Attacker hijack user session

Excessive Data Exposure

Vulnerability Definition:

API vulnerability that exposes more data than allowed to the clients

Impact: Allows unauthorized access to sensitive information, including PII, financial data and business data.

How-to-exploit:

[1] Through misconfigured security settings, weak authentication measures or exposed error messages.

[2] API does not implement fine-grained control over what data is sent to which client.

[3] API sends a standard set of data for all requests, regardless of the specific needs or privileges of the client.

[4] API fails to consider the user's role, permissions, or other context that should dictate what data they can access

Best Practices

- Implement strong access controls
- Use encryption to protect data.
- Implement input validation techniques
- Use proper error-handling techniques
- Implement secure data handling procedures
- Schedule regular security audits

Lack of Resources and Rate Limiting

Vulnerability Definition:

API does not have sufficient resources to handle the volume of requests it receives. Unspecified rate limiting, unable to control the amount of incoming requests to an API within a given timeline

Impact: Can compromise API functionality and data availability, outages.

How-to-exploit:

At risk of Denial of Service (DoS) or Distributed Dos (DDos) attacks

[1] If the server hosting the API does not have enough processing power or memory, it may become overwhelmed by too many requests, it can lead to a breakdown of the entire system.

[2] Unexpected spikes in traffic, whether legitimate or due to an attack, can drain resources. An attacker can flood the API with requests, leading to resource exhaustion and potential service outages.

Best Practices

- Implement rate-limiting techniques
as: rate-limiting headers, caching and use of AWS WAF
- Use token-based HTTP authentication to prevent credential-stuffing attacks

- Enable CAPTCHA for suspicious activities
- Use cloud services or scalable solutions
- Implement caching when appropriate to reduce load
- Set up alerts for unusual patterns that might indicate an attack or a resource bottleneck
- Have plans in place for how the API will degrade gracefully under high load conditions

Broken Object-level Authorization “BOLA”

Vulnerability definition:

API endpoint exposes objects without adequate checks of the user's rights or permissions

Impact: Access to sensitive data

How-to-exploit:

[1] Attackers to manipulate the URLs or the API request to gain access to resources they should not have access to, such as other users' data

Best Practices

- Use Role-based access controls (RBAC) technology
- Properly implement authorization checks at the object level
- Validate all inputs when accessing object data
- Implement proper session management
- Use indirect references or other methods to abstract direct access to objects.
- Implement ACLs that define and enforce who can access which resources.

Broken Function-Level Authorization

Vulnerability definition: users (authenticated or not) can access functionalities that they shouldn't be able to.

Impact: Access to sensitive functions or data

How-to-exploit:

[1] Due misconfigured or weak access controls, actors are able to perform escalated actions, leading to data breach or application hijacking.

[2] Insufficient role management and access controls.

[3] Flawed authentication mechanisms.

[4] Inadequate security checks at the function level.

Best practices

- Implement strict access ctrls to ensure appropriate role-based access to sensitive data
- Use an automated access control mechanism
- Implement regularly scheduled device updates
- Stay current with information and vulnerability feeds and exploit databases.
- Use principles like least privilege and segregation of duties.

Mass Assignment Vulnerabilities

Vulnerability Definition:

Allows an anonymous actor to assign values without validation or filtering. Assigns user-supplied data to object properties without adequate filtering

Impact: Unauthorized access and modification of data can take place this way.

How-to-exploit:

[1] Attackers can exploit mass assignment to gain access to data fields they shouldn't be able to, such as user roles, permissions, or personal data.

[2] An attacker can modify critical data, if these fields are inadvertently exposed

[3] Mass assignment vulnerabilities can be exploited systematically, leading to widespread data breaches or integrity issues.

[4] interact with complex data models, and inadvertently exposing a single attribute

Best Practices:

- Use specialized frameworks that prevent mass-assignment attacks.
- Use a whitelist object binding within the application data service layer
- Use an immutable or read-only objects policy to prevent unnecessary object modification.
- Use role-based access control to limit what different types of users can modify.

Security Misconfiguration

Vulnerability Definition:

Improperly configured systems and software pose risk to API. Security settings are not defined, implemented, or maintained properly, leaving the system vulnerable to attacks

Impact: unauthorized access or modification

How-to-exploit:

[1] Insufficiently secured cryptography protocols, incorrect file permissions configuration and poor endpoint protection.

[2] Misconfigured cloud storage (like AWS S3 buckets) set to public can lead to unauthorized data access.

[3] Having services and ports open that are not needed for the application's functionality can provide additional attack vector

[4] Displaying detailed error messages, including stack traces or database errors, can give attackers insights into the system's architecture

Best Practices

- Follow OWASP secure coding principles and guidelines.
- Enforce strict access controls
- Use a secure configuration management process that reduces the attack surface of the API

- Adherence to best practices in configuration, and keeping up-to-date with security patches and updates

Improper Asset Management

Vulnerability Definition:

Poor asset management or failure to identify vulnerable APIs. Inadequate End of life policies and/or dependency management. Oversight of the various components and resources that make up an API ecosystem.

Impact: Gain access to sensitive data and infrastructure

How-to-exploit:

[1] APIs are deprecated or no longer in use, they need to be properly retired to avoid becoming forgotten entry points for attackers

[2] APIs often rely on external libraries or other APIs. Lack of managing these dependencies can result in vulnerabilities.

[3] Improperly configured APIs can expose sensitive data or become entry points for attacks.

Best Practices

- Use well-documented inventory lists of all physical and virtual devices
- Use a comprehensive asset discovery system to identify vulnerability exposures and risks.
- Regularly test application environments to identify any potential vulnerabilities and gaps in security practices.

Insufficient Logging and Monitoring

Vulnerability Definition:

Lack of insufficient logs, monitoring, observability tools, or recording of activities within an API system.

Impact:

Prevents business from keeping an eye on assets, at risk of unauthorized access. Difficult to detect and prevent data breaches

How-to-exploit:

[1] Without proper logging, it becomes difficult to identify and react to unauthorized access or other security breaches promptly. Activities might go unnoticed until it's too late.

[2] Unable to trace actions back to individual users. Becomes challenging to establish accountability for actions performed within the API.

Best Practices:

- Use an automated logging and monitoring system to keep track of user and client activity
- Use security information and event management (SIEM) tools.
- Identify, analyze and investigate and respond to suspicious activity on the API in real-time via alerts and notifications.

